SAND REPORT

SAND2004-3268
Unlimited Release
Printed June 2004

Teuchos::RefCountPtr Beginner's Guide

An Introduction to the Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++

Roscoe A. Bartlett
Optimization and Uncertainty Estimation

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2004-3268
Unlimited Release
Printed June 2004

Teuchos::RefCountPtr Beginner's Guide

An Introduction to the Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++

Roscoe A. Bartlett
Optimization and Uncertainty Estimation
Sandia National Laboratories*, Albuquerque NM 87185 USA,

Abstract

Dynamic memory management in C++ is one of the most common areas of difficulty and errors for amateur and expert C++ developers alike. The improper use of operator `new` and operator `delete` is arguably the most common cause of incorrect program behavior and segmentation faults in C++ programs. Here we introduce a templated concrete C++ class `Teuchos::RefCountPtr<>`, which is part of the Trilinos tools package `Teuchos`, that combines the concepts of smart pointers and reference counting to build a low-overhead but effective tool for simplifying dynamic memory management in C++. We discuss why the use of raw pointers for memory management, managed through explicit calls to operator `new` and operator `delete`, is so difficult to accomplish without making mistakes and how programs that use raw pointers for memory management can easily be modified to use `RefCountPtr<>`. In addition, explicit calls to operator `delete` is fragile and results in memory leaks in the presents of C++ exceptions. In its most basic usage, `RefCountPtr<>` automatically determines when operator `delete` should be called to free an object allocated with operator `new` and is not fragile in the presents of exceptions. The class also supports more sophisticated use cases as well. This document describes just the most basic usage of `RefCountPtr<>` to allow developers to get started using it right away. However, more detailed information on the design and advanced features of `RefCountPtr<>` is provided by the companion document "Teuchos::RefCountPtr : The Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++" [2].

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Acknowledgments

The author would like to thank Carl Laird, Heidi Thornquist, Mike Heroux and Marzio Sala for comments on earlier drafts of this document.

The format of this report is based on information found in [5].

Contents

Appendix

Teuchos::RefCountPtr Beginner's Guide

An Introduction to the Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++

1 Introduction

The main purpose of this document is to provide a quick-start guide on how to incorporate the reference-counting smart pointer class `Teuchos::RefCountPtr<>` into C++ programs that use dynamic memory allocation and object orientation. This code is included in the Trilinos [4] tools package `Teuchos`. The design of `Teuchos::RefCountPtr<>` is based partly on the interface for `std::auto_ptr<>` and Items 28 and 29 in "More Effective C++" [6]. In short, `RefCountPtr<>` allows one client to dynamically create an object (using operator `new` for instance), pass the object around to other clients that need to access the object and never require any client to explicitly call operator `delete`. The object will (almost magically) be deleted when all of the clients remove their references to the object. In principle, this is very similar to the type of garbage collection that is in languages like Perl and Java. There are some pathological cases (such as the classic problem of circular references, see [6, Item 29, page 212]) where `RefCountPtr<>` will result in a memory leak, but these situations can be avoided through the careful use of `RefCountPtr<>`. However, realizing the potential of hands-off garbage collection with `RefCountPtr<>` requires following some rules. These rules are partially spelled out in the form of commandments in Appendix C.

Note that direct calls to operator `delete` are discouraged in modern C++ programs that are designed to be robust in the presence of C++ exception handing. This is because the raw use of operator `delete` often results in memory leaks when exceptions are thrown. For example, in the code fragment:

```
void someFunction() {
  A *a = new A;
  a->f();
  delete a;
}
```

if an exception is thrown in the function call `a->f()` then the statement `delete a` will never be

executed and a memory leak will have been created. The class `std::auto_ptr<>` was added to the standard C++ library (see [6, Items 9 and 10]) to protect against these types of memory leaks. For example, the rewritten function:

```
void someFunction() {
  std::auto_ptr<A> a(new A);
  a->f();
}
```

is robust in the event of exceptions and no memory leak will occur. However, `std::auto_ptr<>` can not be used to share a resource between two or more clients and therefore is not an answer to the issue of general garbage collection. The class `RefCountPtr<>` not only is robust in the event of exceptions but also implements reference counting and is therefore more general (but admittedly more complex and expensive) than `std::auto_ptr<>`.

The use of `RefCountPtr<>` is critically important in the development and maintenance of large complex object-oriented programs composed of many separately-developed pieces (such as Trilinos). This discussion assumes that the reader has a basic familiarity and some programming experience with C++ and has at least been exposed to the basic concepts of object-oriented programming (good sources include [3] and [7]). Furthermore, the reader should be comfortable with the use of C++ pointers and references.

The appendices contain basic reference material for `RefCountPtr<>`. In many respects, the appendices are the most important contribution of this document. For those readers that like to see the C++ declarations right away, Appendix A contains the C++ declarations for the template class `RefCountPtr<>` and some important associated non-member templated functions. Appendix B is a short reference-card-like quick-start for the use of `RefCountPtr<>`. The quick-start in this appendix shows how to create `RefCountPtr<>` objects from raw C++ pointers, how to represent different forms on constantness, cast from one pointer type to another, access the underlying reference-counted object as well as to associate and manage extra data. Appendix C gives some commandments for the use of `RefCountPtr<>` and reinforces the material in Appendix B. Appendix D gives tables of recommended idioms for how to pass raw C++ objects and `RefCountPtr<>`-wrapped objects to and from functions. More detailed discussions of all of the material in the appendices is contained in the design document for `RefCountPtr<>` [2]. Appendix E gives a listing for an example program that uses raw pointer variables and direct calls to operator `new` and operator `delete` while Appendix F shows a refactoring of this example program to use `RefCountPtr<>`.

Note! Anxious readers are encouraged to jump directly to Appendix E and F to get an idea of what `RefCountPtr<>` is all about. This example, together with the reference material in the appendices, should be enough for semi-experienced C++ developers to start using `RefCountPtr<>` right away.

For less anxious readers, in the following section, we describe why the use of raw C++ pointers and raw calls to operator `new` and especially operator `delete` is difficult to program correctly in even

moderately complex C++ programs. We then discuss the different ways C++ pointers are used in such programs and describe how to refactor these programs to replace some of the raw C++ pointers and raw calls to operator `delete` with `RefCountPtr<>`. In the following discussion we will define *persisting* and *non-persisting* associations and will make a distinction between them (see page 11). `RefCountPtr<>` is recommended for use only with *persisting* associations. The consistent use of `RefCountPtr<>` extends the vocabulary of C++ in helping to distinguish between these two types of relationships. In addition, `RefCountPtr<>` is designed for the memory management of individual objects, not raw C++ arrays of objects. Array allocation and deallocation should be performed using standard C++ containers such as `std::vector<>`, `std::valarray<>` or some other such convenient C++ array class. However, it is quite common to dynamically allocate arrays of `RefCountPtr<>` objects and use `RefCountPtr<>` to manage the lifetime of such array class objects.

2 An example C++ program

The use of object-oriented (OO) programing in C++ is the major motivation for the development of `RefCountPtr<>`. OO programs are characterized by the use of abstract classes (i.e. interfaces) and concrete subclasses (i.e. implementations). In OO programs it is common that the selection of which concrete subclass(es) to use is not known until runtime. The "Abstract Factory" [3] is a popular design pattern that allows the flexible runtime selection of what concrete subclasses to create.

Below we describe a fictitious program that demonstrates some of the typical features of an OO program that uses dynamic memory management in C++. In this simple program, handling memory management using raw C++ pointers and calls to operator `new` and operator `delete` will appear fairly easy but larger more realistic OO programs are much more complicated and it is definitely not easy to do memory management without some help.

2.1 Example C++ program using raw dynamic memory management

One of the predominate features of this example program is the use of the following abstract interface base class `UtilityBase` that defines an interface to provide some useful capability.

```
class UtilityBase {
public:
  virtual void f() const = 0;
};
```

In our example program, `UtilityBase` will have two subclasses where one or the other will be used at runtime.

```
class UtilityA : public UtilityBase {
public:
  void f() const { std::cout<<"\nUtilityA::f() called, this="<<this<<"\n"; }
};

class UtilityB : public UtilityBase {
public:
  void f() const { std::cout<<"\nUtilityB::f() called, this="<<this<<"\n"; }
};
```

In this example program the above implementation functions just print to standard out.

Some of the clients in this program have to create `UtilityBase` objects without knowing exactly what concrete subclasses are being used. This is accomplished through the use of the "Abstract Factory" design pattern [3]. For `UtilityBase`, the abstract factory looks like

```
class UtilityBaseFactory {
public:
  virtual UtilityBase* createUtility() const = 0;
};
```

and has the following factory subclasses for creating `UtilityA` and `UtilityB` objects.

```
class UtilityAFactory : public UtilityBaseFactory {
public:
  UtilityBase* createUtility() const { return new UtilityA(); }
};

class UtilityBFactory : public UtilityBaseFactory {
public:
  UtilityBase* createUtility() const { return new UtilityB(); }
};
```

Now let's assume that our example program has the following client classes.

```
// Simple client with no state
class ClientA {
public:
  void f( const UtilityBase &utility ) const { utility.f(); }
};

// Client that maintains a pointer to a Utility object
class ClientB {
```

```
  UtilityBase *utility_;
public:
  ClientB() : utility_(0) {}
  ~ClientB() { delete utility_; }
  void initialize( UtilityBase *utility ) { utility_ = utility; }
  void g( const ClientA &a ) { a.f(*utility_); }
};

// Client that maintains pointers to UtilityFactory and Utility objects
class ClientC {
  const UtilityBaseFactory *utilityFactory_;
  UtilityBase              *utility_;
  bool                     shareUtility_;
public:
  ClientC( const UtilityBaseFactory *utilityFactory, bool shareUtility )
    :utilityFactory_(utilityFactory)
    ,utility_(utilityFactory->createUtility())
    ,shareUtility_(shareUtility) {}
  ~ClientC() { delete utilityFactory_; delete utility_; }
  void h( ClientB *b ) {
    if( shareUtility_ ) b->initialize(utility_);
    else                b->initialize(utilityFactory_->createUtility());
  }
};
```

The type of logic used in `ClientC` for determining when new objects should be created or when objects should be reused and passed around is common in larger more complicated OO programs.

The above client classes demonstrate two different types of associations between objects: *non-persisting* and *persisting*.

Non-Persisting associations exist only within a single function call and do not extend after the function has finished executing. For example, objects of type `ClientA` and `UtilityBase` have a non-persisting relationship through the function `ClientA::f(const UtilityBase &utility)`. Likewise, objects of type `ClientB` and `ClientA` have a non-persisting association through the function `ClientB::g( const ClientA &a )`.

Persisting associations are where a relationship between two objects exists past a single function call. The most typical kind of persisting association in an OO C++ program is where one object maintains a private pointer data member to another object. For example, persisting associations exist between a `ClientC` object, a `UtilityBaseFactory` and a `UtilityBase` object through the the private C++ pointer data members `ClientC::utilityFactory_` and `ClientC::utility_` respectively. Likewise, a persisting association exists between a `ClientB` object and a `UtilityBase` object through the private pointer data member `ClientB::utility_`.

Persisting relationships are significantly more complex than non-persisting relationships since

a persisting relationship usually implies that some objects must be responsible for the lifetime of other objects. This is never the case in a non-persisting relationship as defined above.

Appendix E shows an example program that uses all of the C++ classes described above. The program in Appendix E has several memory management problems. An astute reader will notice that the `UtilityBaseFactory` created in `main()` gets deleted twice; once in the destructor for the `ClientC` object `c` and again at the end of `main()` in an explicit call to operator `delete`. This problem could be fixed in this program by arbitrating "ownership" of the `UtilityBaseFactory` object to either `main()` or the `ClientC` object, but not both which is the case in Appendix E.

A more difficult memory management problem to catch and fix occurs in the `ClientB` and `ClientC` objects regrading a shared `UtilityBase` object. When `shareUtility` is set to `false` (by the user in the commandline arguments) the objects `b1`, `b2` and `c` each own a pointer to different `UtilityBase` objects and the software will correctly delete each dynamically allocated object using one and only one call to operator `delete` (in the destructors of these classes). However, when `shareUtility` is to set to `true` the objects `b1`, `b2` and `c` will contain pointers to the same `UtilityBase` object and operator `delete` will be called on this shared `UtilityBase` object multiple times when `b1`, `b2` and `c` are destroyed. In this case, it is not so easy to arbitrate ownership of the shared `UtilityBase` object to the `ClientB` or the `ClientC` objects. Logic could be developed in this simple program to insure that ownership was assigned properly but such logic would enlarge the program, complicate maintenance, and would ultimately make the software components less reusable. In more complex programs, trying to dynamically arbitrate ownership at run time is much more difficult and error prone if done manually.

2.2 Refactored example C++ program using `RefCountPtr<>`

Now we describe how `RefCountPtr<>` can be used to greatly simplify dynamic memory management in these types of OO programs. Appendix F shows the refactoring of the program in Appendix E to use `RefCountPtr<>` for all persisting relationships. In general, refactoring software that uses raw C++ pointers to use `RefCountPtr<>` is as simple as replacing the type `T*` with `RefCountPtr<T>`, where `T` is nearly any class or built-in data type.

The first persisting relationship for which `RefCountPtr<>` is used is the relationship between a `UtilityBaseFactory` object and a client that uses it. The refactoring changes the return type of `UtilityBaseFactory::createUtility()` from a raw `UtilityBase*` pointer to a `RefCountPtr<UtilityBase>` object. The new "Abstract Factory" class declarations (assuming that the symbols from the `Teuchos` namespace are in scope so that explicit `Teuchos::` qualification is not necessary) become

```
class UtilityBaseFactory {
public:
  virtual RefCountPtr<UtilityBase> createUtility() const = 0;
```

```
};

class UtilityAFactory : public UtilityBaseFactory {
public:
  RefCountPtr<UtilityBase> createUtility() const { return rcp(new UtilityA()); }
};

class UtilityBFactory : public UtilityBaseFactory {
public:
  RefCountPtr<UtilityBase> createUtility() const { return rcp(new UtilityB()); }
};
```

In addition to the change of the return type, the refactoring also requires that calls to operator `new` be wrapped in calls to the templated function `Teuchos::rcp(...)`.

The refactoring shown in Appendix F does not impact the definition of the class `ClientA` since this class does not have any persisting relationships with any other objects. However, the definitions of the classes `ClientB` and `ClientC` do change and become

```
class ClientB {
  RefCountPtr<UtilityBase> utility_;
public:
  void initialize(const RefCountPtr<UtilityBase> &utility) { utility_=utility; }
  void g( const ClientA &a ) { a.f(*utility_); }
};

class ClientC {
  RefCountPtr<UtilityBaseFactory> utilityFactory_;
  RefCountPtr<UtilityBase>        utility_;
  bool                            shareUtility_;
public:
  ClientC( const RefCountPtr<UtilityBaseFactory> &utilityFactory, bool shareUtility )
    :utilityFactory_(utilityFactory)
    ,utility_(utilityFactory->createUtility())
    ,shareUtility_(shareUtility) {}
  void h( ClientB *b ) {
    if( shareUtility_ ) b->initialize(utility_);
    else                b->initialize(utilityFactory_->createUtility());
  }
};
```

The first thing that one should notice about the refactored `ClientB` and `ClientC` classes is that their destructors are gone. It turns out that the compiler-generated destructors do exactly the correct thing (i.e. call the destructor on the `RefCountPtr<>` data members which in turns calls operator `delete` on the underlying reference-counted object when the reference count goes to zero). The

second thing that one should notice is that the old default constructor `ClientB::ClientB()` which initialized the raw C++ pointer `utility_` to null is no longer needed since `RefCountPtr<>` has a default constructor that does that. A third thing to notice about these refactored client classes is that the `RefCountPtr<>` objects are passed by `const` reference (see Appendix D) and not by value as the corresponding raw pointers where in the original unfactored classes. Passing `RefCountPtr<>` objects by `const` reference yields slightly more efficient code and simplifies stepping through the code in a debugger. For example, a function declared as

```
void someFunction( RefCountPtr<A> a );
```

will always result in the copy constructor for `RefCountPtr<>` being called (and therefore stepped into in a debugger) while this same function declared as:

```
void someFunction( const RefCountPtr<A> &a );
```

will often not require the copy constructor be called (except in cases where an implicit conversion is being performed as described in Appendix B) and thereby easing debugging.

As an aside, note that Appendix D gives recommended idioms for how to pass raw C++ objects and `RefCountPtr<>`-wrapped objects to and from functions in a way that result in function prototypes becoming as self documenting as possible, help to avoid coding errors and increase the readability of C++ code. Also, in addition to the benefit that `RefCountPtr<>` eases dynamic memory management, the selective use of `RefCountPtr<>` and raw C++ object references extends the vocabulary of the C++ language by helping to distinguish between persisting and non-persisting associations. For example, when a one sees a function prototype where an object is passed through a `RefCountPtr<>` such as

```
class SomeClass {
public:
  void someFunction( const RefCountPtr<A> &a );
}
```

one can automatically deduce that “memory” of the `A` object will be retained (through a private `RefCountPtr<A>` data member in `SomeClass` no doubt) and that should automatically alter how the developer plans on calling that function and passing the `A` object. The refactored C++ program in Appendix F provides an example of how the idioms presented in Appendix D are put to use.

3 Additional and advanced features of `RefCountPtr<>`

The use cases for `RefCountPtr<>` described above comprise a large majority of the relavent use cases in most programs, but there there are some other use cases that require additional and more

advanced features. Some of these additional features (the C++ declarations for which are shown in Appendix A) are mentioned below:

1. **Casting**

   `RefCountPtr<>` objects can be casted in a manner similar to casting raw C++ pointers and the same types of conversion rules apply. Analogs of the built-in casts `static_cast<>`, `const_cast<>` and `dynamic_cast<>` are supported by the non-member templated functions `rcp_static_cast<>`, `rcp_const_cast<>` and `rcp_dynamic_cast<>` respectively. See Appendix B for examples of how they are used.

2. **Reference-count information**

   The function `RefCountPtr<>::count()` returns the number of `RefCountPtr<>` objects that point to the underlying reference-counted object. This information can be useful in some cases.

3. **Associating extra data with a reference-counted object**

   There are some more difficult use cases where certain types of information or other objects must be bundled with a reference-counted object and must not be deleted until the reference-counted object is deleted. The non-member templated functions `set_extra_data<>(...)` and `get_extra_data<>(...)` serve this purpose (see item (5) in Appendix B).

4. **Customized deallocation policies**

   The default behavior of `RefCountPtr<>` is to call operator `delete` on reference-counted objects once the reference count goes to zero. While this is the most commonly needed behavior, there are use cases where more specialized dellocation polices are required. For these cases, there is an overloaded form of the templated function `Teuchos::rcp(...)` that takes a templated deallocation policy object that defines how a reference-counted object is deallocated when required.

These features are discussed in detail in the design document [2].

4 Summary

The templated C++ class `RefCountPtr<>` provides a low-overhead option for (almost) automatic memory management in C++. This class has been developed and refined over many years and has been instrumental in improving the quality of software projects that use it consistently (for example see MOOCHO [1]). Careful use of `RefCountPtr<>` eliminates the need to manually call operator `delete` when dynamically allocated objects are no longer needed. Furthermore, it helps to reduce the amount of code that developers have to write. For example, most classes that use `RefCountPtr<>` for dynamically allocated memory do not need developer-supplied destructors. This because

the compiler-generated destructors do the exactly correct thing which is to call destructors on an object's constituent data members. This was demonstrated in the difference between the original and refactored classes `ClientB` and `ClientC` described in Sections 2.1 and 2.2.

The class `RefCountPtr<>` also has advanced features not found in other smart-pointer implementations such as the ability to attach extra data and the customization of the deallocation policy.

References

[1] R. A. Bartlett. *MOOCHO : Multifunctional Object-Oriented arCHitecture for Optimization, User's Guide*. Sandia National Labs, 2003.

[2] Roscoe A. Bartlett. Teuchos::RefCountPtr : The Trilinos smart reference-counted pointer class for (almost) automatic dynamic memory management in C++. Technical Report In preparation, Sandia National Laboratories, 2004.

[3] E. Gamma, R. Helm, R. Johnson, and John Vlissides. *Design Patterns: Elements fo Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[5] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.

[6] S. Meyers. *More Effective C++*. Addison-Wesley, 1996.

[7] B. Stroustrup. *The C++ Programming Language, special edition*. Addison-Wesley, New York, 2000.

A C++ declarations for `RefCountPtr<>`

```
namespace Teuchos {

enum ENull { null };

enum EPrePostDestruction { PRE_DESTROY, POST_DESTROY };

template<class T>
class DeallocDelete { public: void free( T* ptr ) { if(ptr) delete ptr; }  };

template<class T>
class DeallocArrayDelete { public: void free( T* ptr ) { if(ptr) delete [] ptr; }  };

template<class T>
class RefCountPtr {
public:
  typedef T  element_type;
  RefCountPtr( ENull null_arg = null );
  RefCountPtr(const RefCountPtr<T>& r_ptr);
  template<class T2> RefCountPtr(const RefCountPtr<T2>& r_ptr);
  ~RefCountPtr();
  RefCountPtr<T>& operator=(const RefCountPtr<T>& r_ptr);
  T* operator->() const;
  T& operator*() const;
  T* get() const;
  T* release();
  int count() const;
  void set_has_ownership();
  bool has_ownership() const;
  bool shares_resource(const RefCountPtr<T>& r_ptr) const;
private:
  ...
};

template<class T>            RefCountPtr<T>   rcp( T* p );
template<class T>            RefCountPtr<T>   rcp( T* p, bool owns_mem);
template<class T
  ,class Dealloc_T>          RefCountPtr<T>   rcp( T* p, Dealloc_T dealloc, bool owns_mem );
template<class T2, class T1> RefCountPtr<T2>  rcp_implicit_cast(const RefCountPtr<T1>& p1);
template<class T2, class T1> RefCountPtr<T2>  rcp_static_cast(const RefCountPtr<T1>& p1);
template<class T2, class T1> RefCountPtr<T2>  rcp_const_cast(const RefCountPtr<T1>& p1);
template<class T2, class T1> RefCountPtr<T2>  rcp_dynamic_cast(const RefCountPtr<T1>& p1);
template<class T1, class T2> int              set_extra_data( const T1 &extra_data
                                                ,const std::string& name, RefCountPtr<T2> *p
                                                ,bool force_unique = true
                                                ,EPrePostDestruction destroy_when = POST_DESTROY );
template<class T1, class T2> T1&              get_extra_data( RefCountPtr<T2>& p
                                                ,const std::string& name );
template<class T1, class T2> const T1&        get_extra_data( const RefCountPtr<T2>& p
                                                ,const std::string& name );
template<class Dealloc_T
  , class T>                 Dealloc_T&       get_dealloc( RefCountPtr<T>& p );
template<class Dealloc_T
  , class T>                 const Dealloc_T& get_dealloc( const RefCountPtr<T>& p );

}
```

B `RefCountPtr<>` quick-start and reference

This appendix presents a short, but fairly comprehensive, quick-start for the use of `RefCountPtr<>`. The use cases described here should cover the overwhelming majority of the use instances of `RefCountPtr<>` in a typical program.

The following class hierarchy will be used in the C++ examples given below.

```
class A { public: virtual ~A(){} A& operator=(const A&){} virtual void f(){} };
class B1 : virtual public A {};
class B2 : virtual public A {};
class C : virtual public B1, virtual public B2 {};

class D {};
class E : public D {};
```

All of the following code examples used in this appendix are assumed to be in the namespace `Teuchos` or have appropriate `using Teuchos::...` declarations. This removes the need to explicitly use `Teuchos::` to qualify classes, functions and other declarations from the `Teuchos` namespace.

1. **Creation of `RefCountPtr<>` objects**

   (a) **Creating a `RefCountPtr<>` object using `new`**

   ```
   RefCountPtr<C> c_ptr = rcp(new C);
   ```

   (b) **Creating a `RefCountPtr<>` object to an array allocated using `new[n]`**

   ```
   RefCountPtr<C> c_ptr = rcp(new C[n],DeallocArrayDelete<C>(),true);
   ```

   (c) **Initializing a `RefCountPtr<>` object to NULL**

   ```
   RefCountPtr<C> c_ptr;
   ```

   or

   ```
   RefCountPtr<C> c_ptr = null;
   ```

   (d) **Initializing a `RefCountPtr<>` object to an object <u>not</u> allocated with `new`**

   ```
   C              c;
   RefCountPtr<C> c_ptr = rcp(&c,false);
   ```

   (e) **Representing constantness and non-constantness**

      i. **Non-constant pointer to non-constant object**

      ```
      RefCountPtr<C> c_ptr;
      ```

      ii. **Constant pointer to non-constant object**

```
const RefCountPtr<C> c_ptr;
```

iii. **Non-Constant pointer to constant object**

```
RefCountPtr<const C> c_ptr;
```

iv. **Constant pointer to constant object**

```
const RefCountPtr<const C> c_ptr;
```

(f) **Copy constructor (implicit casting)**

```
RefCountPtr<C>       c_ptr  = rcp(new C); // No cast
RefCountPtr<A>       a_ptr  = c_ptr;      // Cast to base class
RefCountPtr<const A> ca_ptr = a_ptr;      // Cast from non-const to const
```

2. **Reinitialization of `RefCountPtr<>` objects (using assignment operator)**

(a) **Resetting from a raw pointer**

```
RefCountPtr<A> a_ptr;
a_ptr = rcp(new C());
```

(b) **Resetting to null**

```
RefCountPtr<A> a_ptr = rcp(new C());
a_ptr = null; // The C object will be deleted here
```

(c) **Assigning from a `RefCountPtr<>` object**

```
RefCountPtr<A> a_ptr1;
RefCountPtr<A> a_ptr2 = rcp(new C());
a_ptr1 = a_ptr2; // Now a_ptr1 and a_ptr2 point to same C object
```

3. **Accessing the reference-counted object**

(a) **Access to object reference (runtime checked)**

```
C &c_ref = *c_ptr;
```

(b) **Access to object pointer (unchecked, may return `NULL`)**

```
C *c_rptr = c_ptr.get();
```

(c) **Access to object pointer (runtime checked, will not return `NULL`)**

```
C *c_rptr = &*c_ptr;
```

(d) **Access of object's member (runtime checked)**

```
c_ptr->f();
```

4. **Casting**

(a) **Implicit casting (see copy constructor above)**

(b) **Casting away `const`**

```
RefCountPtr<const A>  ca_ptr = rcp(new C);
RefCountPtr<A>        a_ptr  = rcp_const_cast<A>(ca_ptr); // cast away const!
```

(c) **Static cast (no runtime check)**

```
RefCountPtr<D>     d_ptr = rcp(new E);
RefCountPtr<E>     e_ptr = rcp_static_cast<E>(d_ptr); // Unchecked, unsafe?
```

(d) **Dynamic cast (runtime checked)**

```
RefCountPtr<A>     a_ptr  = rcp(new C);
RefCountPtr<B1>    b1_ptr = rcp_dynamic_cast<B1>(a_ptr);  // Checked, safe!
RefCountPtr<B2>    b2_ptr = rcp_dynamic_cast<B2>(b1_ptr); // Checked, safe!
RefCountPtr<C>     c_ptr  = rcp_dynamic_cast<C>(b2_ptr);  // Checked, safe!
```

5. **Managing extra data**

(a) **Adding extra data (post destruction of extra data)**

```
set_extra_data(rcp(new B1),"A:B1",&a_ptr);
```

(b) **Adding extra data (pre destruction of extra data)**

```
set_extra_data(rcp(new B1),"A:B1",&a_ptr,PRE_DESTORY);
```

(c) **Retrieving extra data**

```
get_extra_data<RefCountPtr<B1> >(a_ptr,"A:B1")->f();
```

(d) **Resetting extra data**

```
get_extra_data<RefCountPtr<B1> >(a_ptr,"A:B1") = rcp(new C);
```

C Commandments for the use of `RefCountPtr<>`

Here are listed commandments for the use of `RefCountPtr<>`. These commandments reinforce some of the material in the quick-start in Appendix B. The reasoning behind these commandments can be found in the design document for `RefCountPtr<>` [2]. Along with each commandment is one or more anti-commandments stating the negative of the commandment. C++ code fragments are also included to demonstrate each commandment and anti-commandment.

Commandment 1 *Thou shall put a pointer for an object allocated with operator `new` into a `RefCountPtr<>` object only once. The best way to insure this is to call operator `new` directly in a call to `rcp(...)` to create a dynamically allocated object that is to be managed by a `RefCountPtr<>` object (see item (1b) in Appendix B).*

Anti-Commandment 1 *Thou shall never give a raw C++ pointer returned from operator `new` to more than one `RefCountPtr<>` object.*

Example:

```
A *ra_ptr = new C;
RefCountPtr<A> a_ptr1 = rcp(ra_ptr); // Okay
RefCountPtr<A> a_ptr2 = rcp(ra_ptr); // no, No, NO !!!!
```

Anti-Commandment 2 *Thou shall never give a raw C++ pointer to an array of objects returned from operator `new[]` to a `RefCountPtr<>` object.*

Example:

```
RefCountPtr<std::vector<C> > c_array_ptr1 = rcp(new std::vector<C>(N)); // Okay
RefCountPtr<C>               c_array_ptr2 = rcp(new C[N]);             // no, No, NO !!!!
```

Commandment 2 *Thou shall only create a `NULL RefCountPtr<>` object by using the default constructor or by using the `null` enum (and its associated special constructor) (see item (1c) in Appendix B). Trying to assign to `NULL` or `0` will not compile.*

Anti-Commandment 3 *Thou shall not create a `NULL RefCountPtr<>` object using the templated function `rcp(...)` since it is very verbose and complicates maintenance.*

Example:

```
RefCountPtr<A> a_ptr1 = null;          // Yes :-)
RefCountPtr<A> a_ptr2 = rcp<A>(NULL);  // No, too verbose :-(
```

Commandment 3 *Thou shall only pass a raw pointer for an object that is not allocated by operator* `new` *(e.g. allocated on the stack) into a* `RefCountPtr<>` *object by using the templated function* `rcp<T>(T* p, bool owns_mem)` *and setting* `owns_mem` *to* `false` *(see item (1d) in Appendix B).*

Anti-Commandment 4 *Thou shall never pass a pointer for an object not allocated with operator* `new` *into a* `RefCountPtr<>` *object without setting* `owns_mem` *to* `false`*.*

Example:

```
C c;
RefCountPtr<A> a_ptr1 = rcp(&c,false);  // Yes :-)
RefCountPtr<A> a_ptr2 = rcp(&c);        // no, No, NO !!!!
```

Commandment 4 *Thou shalt only cast between* `RefCountPtr<>` *objects using the default copy constructor (for implicit conversions) and the nonmember template functions* `rcp_static_cast<>(...)`, `rcp_const_cast<>(...)` *and* `rcp_dynamic_cast<>(...)` *(see item (4) in Appendix B).*

Anti-Commandment 5 *Thou shall never convert between* `RefCountPtr<>` *objects using raw pointer access.*

Example:

```
RefCountPtr<A>     a_ptr   = rcp(new C);
RefCountPtr<B1>    b1_ptr1 = rcp_dynamic_cast<B1>(a_ptr);         // Yes :-)
RefCountPtr<B1>    b1_ptr2 = rcp(dynamic_cast<B1*>(a_ptr.get())); // no, No, NO !!!
```

D Recommendations for passing objects to and from C++ functions

Below are recommended idioms for passing required[1] and optional[2] arguments into and out of C++ functions for various use cases and different types of objects. These idioms show how to write function arguments prototypes which exploit the C++ language in a way that makes these function prototypes as self documenting as possible, avoid coding errors and increase readability of C++ code. In general, `RefCountPtr<>` objects should be passed and manipulated as though they where raw C++ pointers. The main difference is that while raw C++ pointer objects should generally be passed by value, `RefCountPtr<>` objects should generally be passed by reference for several reasons (see [2] for more details).

Argument purpose	Non-Persisting	Persisting

non-mutable object (required[1])	`S s`
	or
	`const S s`
	or
	`const S &s`
		`const RefCountPtr<const S> &s`

non-mutable object (optional[2])	`const S *s`	`const RefCountPtr<const S> &s`
mutable object	`S *s`	`const RefCountPtr<S> &s`
array of non-mutable objects	`const S s[]`	`const RefCountPtr<const S> s[]`
array of mutable objects	`S s[]`	`const RefCountPtr<S> s[]`

C++ declarations for passing small concrete (i.e. with value semantics) objects to and from functions where `S` is a place holder for an actual built-in or user-defined data type.

Argument purpose	Non-Persisting	Persisting
non-mutable object (required[1])	`const A &a`	`const RefCountPtr<const A> &a`
non-mutable object (optional[2])	`const A *a`	`const RefCountPtr<const A> &a`
mutable object	`A *a`	`const RefCountPtr<A> &a`
array of non-mutable objects	`const A* a[]`	`const RefCountPtr<const A> a[]`
array of mutable objects	`A* a[]`	`const RefCountPtr<A> a[]`

C++ declarations for passing abstract (i.e. with reference or pointer semantics) or large concrete objects to and from functions where `A` is a place holder for an actual abstract C++ base class.

[1] Required arguments must be bound to valid objects (i.e. can not be `NULL`)

[2] Optional arguments may be `NULL` in some cases

E Listing: Example C++ program using raw dynamic memory management

```
#include "example_get_args.hpp"

// Abstract interfaces
class UtilityBase {
public:
  virtual void f() const = 0;
};
class UtilityBaseFactory {
public:
  virtual UtilityBase* createUtility() const = 0;
};

// Concrete implementations
class UtilityA : public UtilityBase {
public:
  void f() const { std::cout<<"\nUtilityA::f() called, this="<<this<<"\n"; }
};
class UtilityB : public UtilityBase {
public:
  void f() const { std::cout<<"\nUtilityB::f() called, this="<<this<<"\n"; }
};
class UtilityAFactory : public UtilityBaseFactory {
public:
  UtilityBase* createUtility() const { return new UtilityA(); }
};
class UtilityBFactory : public UtilityBaseFactory {
public:
  UtilityBase* createUtility() const { return new UtilityB(); }
};

// Client classes
class ClientA {
public:
  void f( const UtilityBase &utility ) const { utility.f(); }
};
class ClientB {
  UtilityBase *utility_;
public:
  ClientB() : utility_(0) {}
  ~ClientB() { delete utility_; }
  void initialize( UtilityBase *utility ) { utility_ = utility; }
  void g( const ClientA &a ) { a.f(*utility_); }
};
class ClientC {
```

```
  const UtilityBaseFactory *utilityFactory_;
  UtilityBase              *utility_;
  bool                     shareUtility_;
public:
  ClientC( const UtilityBaseFactory *utilityFactory, bool shareUtility )
    :utilityFactory_(utilityFactory)
    ,utility_(utilityFactory->createUtility())
    ,shareUtility_(shareUtility) {}
  ~ClientC() { delete utilityFactory_; delete utility_; }
  void h( ClientB *b ) {
    if( shareUtility_ ) b->initialize(utility_);
    else                b->initialize(utilityFactory_->createUtility());
  }
};

// Main program
int main( int argc, char* argv[] )
{
  // Read options from the commandline
  bool useA, shareUtility;
  example_get_args(argc,argv,&useA,&shareUtility);
  // Create factory
  UtilityBaseFactory *utilityFactory = 0;
  if(useA) utilityFactory = new UtilityAFactory();
  else     utilityFactory = new UtilityBFactory();
  // Create cleints
  ClientA a;
  ClientB b1, b2;
  ClientC c(utilityFactory,shareUtility);
  // Do some stuff
  c.h(&b1);
  c.h(&b2);
  b1.g(a);
  b2.g(a);
  // Cleanup memory
  delete utilityFactory;
}
```

F Listing: Refactored example C++ program using `RefCountPtr<>`

```
#include "Teuchos_RefCountPtr.hpp"
#include "example_get_args.hpp"

// Inject symbols for RefCountPtr so we don't need Teuchos:: qualification
using Teuchos::RefCountPtr;
using Teuchos::rcp;

// Abstract interfaces
class UtilityBase {
public:
  virtual void f() const = 0;
};
class UtilityBaseFactory {
public:
  virtual RefCountPtr<UtilityBase> createUtility() const = 0;
};

// Concrete implementations
class UtilityA : public UtilityBase {
public:
  void f() const { std::cout<<"\nUtilityA::f() called, this="<<this<<"\n"; }
};
class UtilityB : public UtilityBase {
public:
  void f() const { std::cout<<"\nUtilityB::f() called, this="<<this<<"\n"; }
};
class UtilityAFactory : public UtilityBaseFactory {
public:
  RefCountPtr<UtilityBase> createUtility() const { return rcp(new UtilityA()); }
};
class UtilityBFactory : public UtilityBaseFactory {
public:
  RefCountPtr<UtilityBase> createUtility() const { return rcp(new UtilityB()); }
};

// Client classes
class ClientA {
public:
  void f( const UtilityBase &utility ) const { utility.f(); }
};
class ClientB {
  RefCountPtr<UtilityBase> utility_;
public:
  void initialize(const RefCountPtr<UtilityBase> &utility) { utility_=utility; }
  void g( const ClientA &a ) { a.f(*utility_); }
```

```
};
class ClientC {
  RefCountPtr<UtilityBaseFactory> utilityFactory_;
  RefCountPtr<UtilityBase>        utility_;
  bool                            shareUtility_;
public:
  ClientC( const RefCountPtr<UtilityBaseFactory> &utilityFactory, bool shareUtility )
    :utilityFactory_(utilityFactory)
    ,utility_(utilityFactory->createUtility())
    ,shareUtility_(shareUtility) {}
  void h( ClientB *b ) {
    if( shareUtility_ ) b->initialize(utility_);
    else                b->initialize(utilityFactory_->createUtility());
  }
};

// Main program
int main( int argc, char* argv[] )
{
  // Read options from the commandline
  bool useA, shareUtility;
  example_get_args(argc,argv,&useA,&shareUtility);
  // Create factory
  RefCountPtr<UtilityBaseFactory> utilityFactory;
  if(useA) utilityFactory = rcp(new UtilityAFactory());
  else     utilityFactory = rcp(new UtilityBFactory());
  // Create cleints
  ClientA a;
  ClientB b1, b2;
  ClientC c(utilityFactory,shareUtility);
  // Do some stuff
  c.h(&b1);
  c.h(&b2);
  b1.g(a);
  b2.g(a);
}
```

DISTRIBUTION:

1 Carl Laird
Department Chemical Engineering
Carnegie Mellon University
5000 Forms Ave.
Pittsburgh, PA 15213

1 Matthias Heinkenschloss
Department of Computational and Applied Mathematics
MS 134 Rice University
6100 S. Main Street
Houston, TX 77005-1892

1 Bill Symes
Department of Computational and Applied Mathematics
MS 134 Rice University
6100 S. Main Street
Houston, TX 77005-1892

1 Tony Padula
Department of Computational and Applied Mathematics
MS 134 Rice University
6100 S. Main Street
Houston, TX 77005-1892

1 Mark Gockenbach
Department of Mathematical Sciences
Michigan Technological University
1400 Townsend Drive
Houghton, Michigan 49931-1295, U.S.A.

1 Paul Sexton
Box 1560
St. John's University
Collegeville, MN 56321

1 MS 0370
Scott Mitchell, 9211

1 MS 0370
David Gay, 9211

5 MS 0370
Roscoe Bartlett, 9211

1 MS 0370
Scott Collis, 9211

1 MS 0370
Bart van Bloemen Waanders, 9211

1 MS 0370
Mike Eldred, 9211

1 MS 0370
Laura Swiler, 9211

1 MS 9159
Mark Adams, 9214

1 MS 1110
Pavel Bochev, 9214

1 MS 1110
Todd Coffey, 9214

1 MS 1110
David Day, 9214

1 MS 1110
John Delaurentis, 9214

1 MS 1110
Michael Heroux, 9214

1 MS 1110
Ulrich Hetmaniuk, 9214

1 MS 9217
Jonathan Hu, 9214

1 MS 1110
Richard Lehoucq, 9214

1 MS 1110
Louis Romero, 9214

1 MS 1110
David Ropp, 9214

1 MS 1110
Mazio Sala, 9214

1 MS 1110
Kendall Stanley, 9214

1 MS 1110
Heidi Thornquist, 9214

1 MS 9217
Raymond Tuminaro, 9214

1 MS 1110
James Willenbring, 9214

1 MS 1110
William Hart, 9215

1 MS 1110
Erik Boman, 9215

1 MS 9159
Paul Boggs, 8962

1 MS 9159
Kevin Long, 8962

1 MS 9159
Patricia Hough, 8962

1 MS 9159
Tamara Kolda, 8962

1 MS 9159
Monica Martinez-Canales, 8962

1 MS 9159
Pamela Williams, 8962

1 MS 9159
Victoria Howle, 8962

1 MS 0316
Eric Keiter, 9233

1 MS 0316
Scott Hutchinson, 9233

1 MS 0316
Robert Hoekstra, 9233

1 MS 0316
Curt Ober, 9233

1 MS 0316
Tom Smith, 9233

1 MS 0316
Russel Hooper, 9233

1 MS 0382
Carter Edwards, 9143

1 MS 0382
James Stewart, 9143

1 MS 0316
Alan Williams, 9143

1 MS 0617
Ricard Drake, 9231

1 MS 0316
Roger Pawlowski, 9233

1 MS 0316
Eric Phipps, 9233

1 MS 1110
Andrew Salinger, 9233

1 MS 1110
Brett Bader, 9233

1 MS 0316
Gary Hennigan, 9233

1 MS 9018
Central Technical Files, 8945-1

2 MS 0899
Technical Library, 9610

2 MS 0612
Review & Approval Desk, 4916